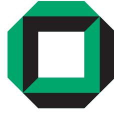

Universität Karlsruhe (TH)
Fakultät für Informatik
System Architecture Group

Dominik Vallendor

Study Thesis

Service Oriented Message Routing for the Structured Overlay Network Igor

Tutor: Kendy Kutzner
Registration date: March 1th, 2007
Submission date: July 10th, 2007

Statement of Authorship

I hereby certify that this study thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged.

Karlsruhe, July 10$^{\text{th}}$, 2007

Contents

1 Introduction **7**

2 Analysis of existing Peer-to-Peer Systems **9**
2.1 Overview of Peer-to-Peer Systems 9
2.2 Examples of structured Peer-to-Peer Systems 10
2.3 Igor 11

3 Design of Service Orientation in Igor **13**
3.1 Challenge 13
3.2 Proposed Solution 13

4 Implementation **17**
4.1 Plugin Concept 17
4.2 Implementation of Policies 18
4.3 Messages in the Igor Network 19
4.4 Message Handling 21
4.5 Periodic Events 23
4.6 Testing 23

5 Abstract **25**
5.1 Conclusions 25
5.2 Further Work 25

Chapter 1

Introduction

This study work deals with the structured overlay network Igor, developed by the System Architecture Group at the University of Karlsruhe.

In the structured overlay network, Igor runs as a daemon on every node which is part of it. Every node has its own node identifier (ID). Dependent on this node ID, the peers know, if they are responsible for a message or how the message has to be routed to its destination. Different applications, like the distributed video recorder Videgor [Fuh07] or a distributed file system may register at the Igor daemon and use it for its purpose. The applications do not need to pay attention to the route their messages walk and by which node they are processed.

Every application program supports a special service. This service has its own service ID, determined from the name of the service. Originally in Igor, every node in the network has to support every application, thus every existing application has to be started with the Igor daemon. If the application and thus the handler for the service is not started with the Igor instance, messages may be lost. If a node is responsible for a certain message (defined by the metric of the Igor network), but the responsible application is not running, the message will be dropped by the daemon.

This behaviour is not practicable. The goal of this work is to change the routing implementation of Igor, so that messages are routed only to those nodes, which handle the service of the message. Therefore the nodes in the network need to have the possibility to determine, which node supports which service. This information has to be held in a distributed manner, safe, robust and quickly available in the network.

Chapter 2

Analysis of existing Peer-to-Peer Systems

2.1 Overview of Peer-to-Peer Systems

According to [SW05] Peer-to-Peer (P2P) systems are classified into the following types:

TYPE	P2P TYPE	DESCRIPTION	EXAPMPLE
Centralized P2P	unstructured	Only one central entity exits, which is neccessary to provide the service.	Napster
Pure P2P	unstructured	No central entity exists.	Freenet, Gnutella 0.4
Hybrid P2P	unstructured	Central entities exist, but these entities are dynamicaly selected out of all peers.	Gnutella 0.6
	structured	Structured Peer-to-Peer systems are based on dynamic hash tables	Chord, CAN, Pastry, Kademlia

A general challenge in P2P is to find out, where it is reasonable to store data. It is also important to find a certain data item in a distributed system, especially in structured overlay networks without any central entity, and it has to be done efficiently.

Early approaches

Early approaches had many drawbacks. One of them is the principle to use centralized servers. The search complexity is $O(1)$, but it has has drawbacks in scalability and availability. The storage capacity, which is necessary for the central server is $O(N)$, which increases with every node N and will become very large for a great number of nodes.

An other principle is flooding search. The search text is sent by a node to all connected nodes which continue sending them to the other nodes until a certain hop count is exceeded. This aproach does not scale well, because the bandwidth used for these messages is very high and cannot be used to retrieve profitable data. On the one hand the

search complexity is $O(N^2)$ or higher. On the other hand the storage costs are only $O(1)$ because the information is only stored on the nodes providing the data.

Distributed Hash Tables (DHTs)

A big improvement in Peer-to-Peer systems came with the introduction of Distributed Hash Tables. Each data item is assigned a unique value (ID) from the address space. This ID is typically calculated from the data itself using an hash algorithm like SHA-1, published in [Pub95]. The Address Space of the IDs is typically 0 to $2^n - 1$. Each node is responsible for specific data items. If a node receives a message for a destination ID, which it is not responsible for, it forwards the message. This process is repeated until the destination node is found. The communication overhead can be $O(\log N)$ and the state information per node scales with $O(\log N)$ (see [SW05]).

2.2 Examples of structured Peer-to-Peer Systems

Chord

Chord is a simple and effective P2P-system, published by Stoica et al. in [SMK+01]. It is based on Distributed Hash Tables with an address space of n. The nodes in the chord network are organized in a circle, ordered by the IDs of the nodes. Each data item is managed by the node whose ID is greater than or equal to the ID of the data item. This node is called the successor of the ID. Each node N in the chord network has a so-called finger table with $\log n$ entries. Each entry points to the successor of $N + 2^l$, where l are the integers ranging from 0 to n. The node opens persistent connections to these successors. Routing is now easily done by forwarding a query to the node closest to the destination. Stoica et al. showed that the average lookup requires only $\frac{1}{2}\log(N)$ steps.

Pastry

Pastry was published by Rowstron and Druschel in [RD01]. In Pastry a network is built, in which nodes and data items have l-bit identifiers. l is typically 128. In contrast to Chord, a data item is located on the node, whose ID is numerically closest. This way it is possible that two nodes are responsible for the same data item if the distance to both node IDs is equal. Every pastry node saves a *routing table*, a *leaf set* and a *neighborhood set*. The *leaf set* contains nodes which are close in the identifier space. The *neighborhood set* contains nodes which are close in the underlying network. Pastry's identifiers are strings of digits to the base 2^b. b is typically chosen to be 4. The *routing table* contains $\frac{l}{r}$ rows with $2^b - 1$ entries per row. In row i the node n stores the identities of other nodes in the network, whose node IDs share an i-digit prefix with n but differ in digit n itself. This way, Pastry nodes have, similiar to Chord nodes, a coarse-grained table of other nodes. The size of the routing information increases with the proximity of other nodes in the identifier space. A lookup in Pastry network requires $\log_{2^b}(N)$ steps.

Content Addressable Network (CAN)

CAN has been published by Ratnasamy et al. in [RFH+01]. Its characteristic is to have a d-dimensional identifier space. Each data item is identified by this d-dimensional identifier, e.g. $< x, y, z >$ for $d = 3$. According to this the geometrical representation of a Content Addressable Network is a d-torus. The identifier space in CAN is partitioned among the participating nodes. Each node owns a zone which is a part of the identifier space. Nodes only store information about their immediate neighbors. If the identifier space is divided into zones of equal size, each node has $2d$ neighbors. In order to gain a qualified identifier for each data item, a uniform hash function is applied. For example, a hash value is calculated from the data item and divided in different segments, which represent the coordinates in the network. Routing is very simple, because it requires only knowledge of a node's immediate neighbors. A query message is forwarded by each node following a straight line in the cartesian identifier space. This results in an average of $O(\frac{d}{4}n^{\frac{1}{d}})$ routing steps.

Kademlia

In the year 2002, Maymounkov and Mazieres presented Kademlia ([MM02]). Its main goal is an improvement of the Pastry algorithm. Pastry's routing metric (identifier prefix length) does not always correspond to the numeric closeness of identifiers, which complicates routing. Therefore Kademlia uses a XOR (bit-wise exclusive OR) metric for measuring the distance between two identifiers. This treats the identifiers of nodes as the leaves of a binary tree and implies a routing latency of $O(\log(N))$ hops on average.

Lanes (DIANE project)

Lanes has first been published in [KKRO03] and the work on it is part of the DIANE-project. DIANE stands for *Dienste in Ad Hoc Netzen*, which means services in ad-hoc networks. The name Lanes derives from the fact that a two-dimensional overlay strucure is built, which divides nodes in lanes. The network is used for semantics-based service discovery. This structure is similar, but not less strict than the one used in CAN. Every node in a lane knows its neighbour nodes, excluding the nodes at the border of the lane. Nodes within a lane communicate via unicast. Service advertisments are propagated in this first dimension. Every node in a lane knows all services which are offered within this lane. Nodes are not aware about the services in other lanes. Nodes between different lanes communicate via anycast. Service requests are distributed in this other dimension.

2.3 Igor

Igor is a Chord-like overlay network, written in C++. The Igor program can be compiled and run as a daemon on Unix systems. Normally a Igor daemon runs on every computer, participating in the Igor network. But also several Igor daemons may be run on different network ports of one computer (e.g. for testing purpose). In this case every daemon behaves as an independent node (like running on a different computer). The ID of every node is chosen randomly at the start of the daemon but may also be specified by configuration. Connections between the individual nodes are established using TCP

connections. For bootstrapping, which is needed at startup of a new daemon, the daemon has to know at least one transport address (Internet-Address/Port) of an other Igor node, which it obtains from an entry in the configuration file.

Igor only provides functionality to route messages to other (responsible) nodes. It does not support any other functions which could be meaningful for end users. These functions, for instance the video recorder Videgor, are provided by special applications which connect to the Igor daemon and typically run on the same machine but does not have to. This concept makes Igor more flexible than other Peer-to-Peer programs.

Chapter 3

Design of Service Orientation in Igor

3.1 Challenge

Well known Peer-to-peer systems only support one special service (e.g. file sharing, distributed video recorder, distributed filesystem, etc.). But if the system ought to support more than only one service, like Igor, this leads to problems.

If all nodes support the same number and types of services, everything is alright, but it also may be possible that nodes support different types of services. Assuming a node in the overlay network sends a service request to another node in the network and this node does not know how to handle this request, it may get lost.

The requesting node has to know which other node supports the reuwested service and is responsible. If there are a big number of nodes N and a lot of services K, this may result in a big challenge and a bottleneck for the whole network.

3.2 Proposed Solution

According to [XYW05] the problem is solved by using multiple interconnected overlay networks.

First, a so-called *attribute overlay network (AON)* is used for management. Every node is part of this attribute overlay network. The AON is used to find and to announce services, which are supported in the whole system. Secondly, a network called *value overlay network (VON)* is used for linking the nodes together, which provide the same service. A node may be member in multiple *value overlay networks*. Figure 3.1 shows the main structure of the networks used in [XYW05].

Actual state of the Igor system is an overlay network with the abilities of standard DHT-based overlay networks. Nodes may join the network and search in the network for data items. This existent network is now used as the *attribute overlay network*.

After a node has joined the *AON*, it searches for its provided services by IDs, determined from the name of the service. Thus a node is used, which is responsible for a certain ID and the service respectively. This node is called the *gateway server*.

Every node in the system has to implement the functions for acting as a *gateway server*. The *gateway server* administers a list of nodes which provides the service(s) the *gateway server* is responsible for. This list does not need to contain the whole information

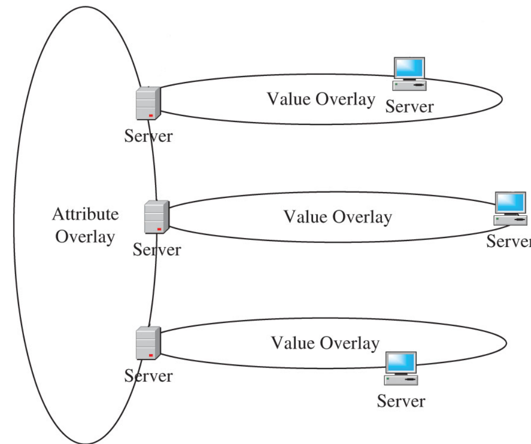

Figure 3.1: Attribute overlay network / Value overlay networks [XYW05]

about all nodes which provide the service. A node that wants to connect a *VON* does not need to know all nodes of the *VON*. It only has to know at least one accessible node in the *VON*. Because of this it should be sufficient, if this list has a finite length. The *gateway server* has to manage a separate list for every service it is responsible for.

If a *gateway server* is asked for a certain service, it returns this list or a part of it. The new node which is joining the network, now tries to enter the *value overlay network* by connecting to one of the nodes which are noted on the list from the *gateway server*. If this bootstrapping is successful, the node is then part of the *value overlay network* and sends an acknowledge message to the *gateway server*. The *gateway server* is now also aware of the node taking part in the *VON* and memorizes this information in its list of nodes. Possibly, old entries in the list are dropped. Also this function will be optimized using a caching mechanism.

In contrast to [XYW05] the *gateway server* himself may also be part of the *value overlay networks*, but this is not a requirement though. After proceeding with the announcement and joining steps for all services provided by the node, it is then member of one or multiple *value overlay networks* in which it may act as in all other overlay networks. Queries concerning a certain service are only used in the *VON*s which are responsible for the respective service.

General considerations

The main question is, how routing of messages for specific services are to be handled. There are two different possiblities of how this can be done:

1. route messages only in the *value overlay network* the message belongs to
2. route messages via all connected nodes in the *attribute overlay network*

Both possibilities have advantages and disadvantages.

The first solution is simple to implement. Also the total number of nodes in the *VON* is smaller than the number of nodes in the whole network. That is why the average hop count for a message to reach its destination is smaller than the hop count with respect

to the whole network and message delivery is faster. The message may take a long time to reach its destination because a disadvantage of this idea is that the nodes in the *VON* are possibly slow.

The second solution should be much faster in practice because a node has the possibility to choose a fast connection from its list of connections to route a message. But the average hop count will be higher. The problem with this solution is that a node outside a *VON* which has received a message, has to find out wheter it is necessary to route a message back into the *VON* or route it forward in the *AON*. If the node outside the *VON* does not know where to find the *VON*, it has to perform a possibly costly lookup. Also the message has to be routed a second time through the whole network if the message has been routed too far. If this happens, routing via all connected nodes will be much slower than routing only inside the *VON*.

For achieving good results, a compromise between both possibilities has to be found. Finding a good compromise may be difficult and is therefore not subject of this study thesis. Because of the more simple and more secure implementation, a message routing via the *VON* will be illustrated in this paper. The plugin structure of Igor will make it possbile to enhance the behavior of the routing in the future.

Caching

A service which is very popular may cause a bottleneck for the whole system at the *gateway server* and the nodes directly connected to it. If a large number of nodes exist which all support the same service, every network join will result in a message to one explicit *gateway server*. This *gateway server* may be overloaded very quickly. To avoid this problem, possibilities to spread the information about nodes in a *value overlay network* to other participants have to be found, whitout bothering the responsible *gateway server*. This can easily be achieved by caching the information about specific services in any nodes of the *AON* which do not have to be the *gateway server*.

Every node manages a list of services and corresponding information called *service cache list*. This list is updated by notification messages which are periodically sent and while joining a *VON*. The messages contains a tuple of the node identifiers and transport addresses, called *node references*. These notifications are sent to the *gateway servers* and normally would pass a lot of other nodes on their way. The messages are not directly forwarded to the *gateway server*. *Node references* are stored in the *service cache list*. In this list the *node references* are collected and will be forwarded after a specifiable time t_1. As a result, network overload will be avoided. After a specifiable time t_2 which is much longer than t_1, an entry will be deleted if it has not been retransmitted in the meantime. Because of this soft-state approach the information about nodes will not become outdated.

Moreover this mechanism has the advantage that popular services are cached much more than unpopular services. This increases performance a lot.

It would also be possible to simply send a service announce message while joining the *VON* and to not expire the entries in the *service cache list* at all or delete it after a much longer time. The advantage of this hard-state approach would be to reduce the total number of messages and traffic in the network. But the disadvantage would be, that the reduced flexibility can cause stability issues for the whole network.

Chapter 4

Implementation

4.1 Plugin Concept

The program code of Igor is written in C++ and is structured in approximately 50 classes. Originally Igor has had a more monolithic code. But while working on this study thesis, the program was transformed and extended into plugins by Axel Sanwald, team member of the System Architecture Group.

The reason to do this was to prepare Igor to be easily extensible in the future. The expansion may lead to a problem with the code of Igor becoming very complex. That is why the code has been split into different parts.

Today four different kinds of plugins exist. These are:

1. cMessagePlugin
2. cPeekPlugin
3. cPolicyPlugin
4. cTaskPlugin

Every type of plugin is implemented in its own class which is derived from the base class *cPlugin*. Figure 4.1 gives a overview of all plugins, which are currently implemented in Igor.

The *cMessagePlugin* is used for handling messages, which are received by a node. A plugin may register a handler for one or multiple message types.

The *cPeekPlugin* can be used for getting informed about incoming and outgoing messages. This may be used for statistic. But no processing is performed by the plugins of the *cPeekPlugin* type.

The *cPolicyPlugin* plugins are used to decide how a message will be routed. Also the plugin will be called every time Igor wants to drop or open a new connection. Every plugin of the *cPolicyPlugin* type has to assign values from 0.0 to 1.0 to the (potential) connections. After calling all plugins, Igor decides which connections will be dropped or opened or how a message will be routed. The *cPolicyPlugin* is the most important of the plugins because it contains functions to manage the CHORD-like network.

The *cTaskPlugin* can be used to trigger events at a certain time. For example this can be used for calling periodic functions.

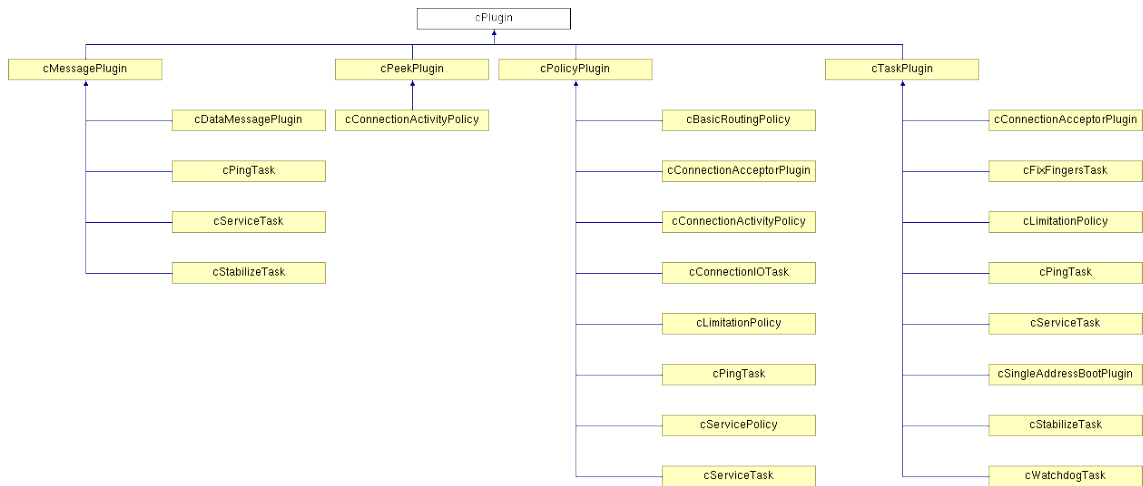

Figure 4.1: class cPlugin and derived plugins

In order to implement services, a plugin has to be written which extends the classes of type *cMessagePlugin*, *cPolicyPlugin* and *cTaskPlugin*. These new classes are called *cServiceTask* (derived from *cMessagePlugin* and *cTaskPlugin*) and *cServicePolicy*, which is derived from *cPolicyPlugin*.

An other important part of the Igor plugin concept is implemented in a class called *cCone* which contains informations about the connections that an Igor instance keeps to other nodes. The *cCone* class is derived from multiple classes. These classes are provided by the plugins and contain informations and functions which the plugins need to work. The new service plugin also expands *cCone* by a class called *cConnectionServiceCone*.

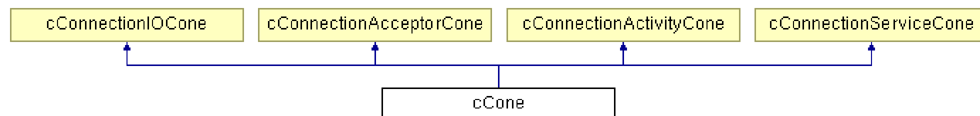

Figure 4.2: class cCone and the classes, it is derived from

Also a new independent class *cServiceCacheList* was built, which implements the *service cache list*.

4.2 Implementation of Policies

In Igor, modules, called policies, are used to determine, whether connections to other nodes are opened or closed. Policies also decide how messages are routed. Every module has to use its own metric depending on the module requirements. For introducing services in Igor, it is necessary to write an own service module. Functions that have to be implemented are:

FUNCTION	WHAT TO DO
EvalPotentialConnection	Walk through the list of own services by walking through the list of client connections and test whether the potential new connection supports one ore more of this services. If no service is supported, the connection should not be used and 0 will be returned. As many as possible services should be supported. If one ore more services are supported, 1 will be returned.
EvalForConnectionDrop	Also walk through the list of own services and decide how useful the connection is. If a connection is not useful any more for any application, this information will be returned to the plugin management which may drop the connection.
EvalForRouting	Messages should not be routed to nodes which do not support the service type of the messages. In case the messages are routed to such a node, this may lead to serious problems like infinite loops. In order to avoid this, the services of the message will be compared to the list of connections. Every connection that supportes the service is evaluated as 1. Every other connection is evaluated as 0.

4.3 Messages in the Igor Network

Igor supports different types of messages to communicate with other nodes. These messages are:

1. msg_wnl, msg_rl (getting new links to other nodes)
2. msg_ping (inform a node about own presence)
3. msg_data (sending a message for an application)
4. msg_getinfo, msg_info (for debugging)

Furthermore there are messages defined for applications to register and unregister at Igor. But functions to handle these messages were not implemented in the new plugin version of Igor and had to be implemented for supporting services. These functions are:

1. msg_register (inform Igor about a new application)
2. msg_register_reply (answer a message)
3. msg_unregister (inform Igor about a disconnecting application)

For implementing support for services in Igor, four new messages had to be introduced:

1. msg_servicelist
2. msg_announce_service
3. msg_want_service
4. msg_want_service_reply

msg_servicelist

To inform directly connected nodes about the services supported by the node, a *msg_servicelist* is sent. The *msg_servicelist* message includes a list of all services which the node supports. Typically, this message will be sent if an application connects or disconnects from a Igor instance. For every connection, every node holds a list of all services, that the connected node supports. This is done in the *cConnectionServiceCone* class. A node does typically not support a very large number of different services. Because of this, there are no special messages for advertisment single services. The *msg_servicelist* message will not grow too much.

msg_announce_service

In contrast to *msg_servicelist*, the *msg_announce_service* is sent to the *gateway server* to inform other, not directly connected nodes about supported services. This message is sent per service and may contain one or more nodes, which support this service. If a node supports a new service, it has to announce this to the network by sending a *msg_announce_service*. The message is also resent in defined intervals. That message is routed to the *gateway server*. The *gateway servers* for different services are typically different nodes, so the messages has to be routed over different connections. That is why it does not make much sense to send information about more than one service in one *msg_announce_service* message and it is not designed. Also because of the persistent connections to connected nodes it is not a lot of more overhead to send multiple messages instead of only one message.

msg_want_service and msg_want_service_reply

If a node wants to find one or more other nodes which support a certain service, e.g. for connecting to the *VON*, it has to ask for it in the network. This is done by sending a message of type *msg_want_service*. The node asks for a special service and a number of node references it wants to have for this service. If another node can answer this message, it sends a *msg_want_service_reply* message back to the inquiring node. The *msg_want_service_reply* message contains the ID of the service and the requested node references asked for.

Structure of service messages

The following table represents the structure of the new messages in addition to standard information like the information about the node which is sending or receiving the message.

MESSAGE	STRUCTURE
msg_servicelist	ServiceID ServiceID ServiceID ...
msg_announce_service	ServiceID Node; Transportaddress Node; Transportaddress Node; Transportaddress ...
msg_want_service	Number of nodes wanted ServiceID
msg_want_service_reply	ServiceID Node; Transportaddress Node; Transportaddress Node; Transportaddress ...

4.4 Message Handling

Plugins may define new types of messages. But then, because service implementation is an integral part of Igor, every node in the *attribute overlay network* has to support the above-named new messages. The message are routed to the responsible node in the network. The responsible node is determined by its ID, e.g. the ID of a service by sending *msg_announce_service* messages.

What happens exactly if a message is received, is explained in the following for each type of message.

msg_register

If this message is received by a node, it has to check whether it was sent by an application, because only application are allowed to send this type of messages. In case the message was received from a peer node, something went wrong and the message is discarded. If the message is sent by an application, but the service of the application, is already registered (by another application), the message is also discard, but the application will be informed about the successful oder unsuccessful registration of the service. In case of successful registration, the new service will be saved in the *cConnectionServiceCone* structure. Also, a new *msg_servicelist* message will be generated and sent to all connected nodes to inform them about the new service. After this, the *msg_register* message will be deleted. The handling function for *msg_register* announces the service to the whole network by sending *msg_announce_service* messages. This is also done in periodical intervals to inform all other nodes, that the the supported services are still available.

msg_unregister

The handling of msg_unregister is much simpler. It is also tested, whether the message was sent by the correct type of connection and then the service will be deleted. Also a new *msg_servicelist* message will be sent to inform about the deleted service. The *msg_unregister* message will be deleted as well. Some connections to other nodes could become useless, if the node does not support any of their services any more. But the connections will not become deleted automatically, because they could be in use by other plugins. The plugin management tries to delete useless connections on its own by querying all plugins periodically, and checking if a connection is still needed.

msg_servicelist

If the message is received from a node which is not connected, the message is discarded. If a node receives the *msg_servicelist* message, it replaces the services in the appropriate list. After this, the message will be deleted.

msg_announce_service

If a node receives a message of this type, it appends the node reference for the given service, provided in the message, to the *service cache list*. Then the message will also be deleted.

msg_want_service

The node tries to answer this message. It looks up the *service cache list*, if it has enough entries to do so. If it does, it will generate a reply message of type *msg_want_service_reply* with all the node references found and will route it back to the origin. If it does not know enough node references to answer the message by its own, it will route the message forward to the *gateway server*. Then the next node will have to answer the message or will also forward it to the *gateway server*. If the message is received by the *gateway server*, it will be answered, regardless whether this node has enough entries to complete the full request. In this case of emergency, it will send all node references it knows. A node which is not the *gateway server* and which cannot answer the *msg_want_service* on its own, does not send parts of the answer (e.g. only 3 of 10 wanted nodes references). It could do so and would let the next node send the remaining 7 entries. But there would be a good chance that duplicated entries would be sent. In this case, the inquiring node would possible only receive 3 different node references. After forwarding or answering the *msg_want_service* message, it will be deleted.

msg_want_service_reply

If the node receives a message of type *msg_want_service_reply*, it will accept it, regardless whether it has sent a *msg_want_service* message before or not. If it receives the message, it tries to find out by asking all policy plugins (also the own service policy), if the provided node references are good or not. It does not matter, whether the receiving host did not send a *msg_want_service* before, because the node references in the *msg_want_service_reply* message could be useful anyway. But it does matter, how many nodes are known for the

given service. If the node knows enough other nodes with this service, it does not handle the message. Otherwise it tries to connect to the provided nodes by sending a reference suggestion to the plugin management. There all other plugins can decide, if they agree in opening a new connection to this node. By opening the connections, the node will become part of the *value overlay network* or stabilize its position in it. Also in this case, the *msg_want_service_reply* mesage will be deleted.

4.5 Periodic Events

For stabilizing the *value overlay network* and to purge internal structures, some things are done in periodical intervals.

First, a msg_servicelist message is sent to all other connected nodes to inform them about all supported services.

Furthermore it is checked, if the node has at least two connections to other nodes for each supported service. Igor is a CHORD-like network and so it needs at minimum of two connections for correct handling. Every service builds its own network, the *value overlay network*, so for every service these two connections are neccessary. If the node does not have enough connections, it tries to set up new ones by sending *msg_want_service* messages.

Furthermore, the *service cache list* is cleaned of stale node reference entries. Then the other nodes are informed about the remaining node references by sending *msg_announce_service* messages to the *gateway servers.*

4.6 Testing

To prove the correct behaviour of the Igor daemon with service support, some tests were conducted. In order to do so, a test application has been developed. The test application is an Igor client program which connects to a given Igor instance. The program supports three different modes:

1. receiving messages for a given service
2. sending a message with a given service ID to a given ID.
3. send messages with a given service to 50 different nodes and then go into receive mode for this service.

Testing was done by starting multiple instances of Igor and connecting different testing applications to them. A few scripts have been used to run the Igor daemons and applications. Once a message for a given service was sent it could be proved that this message only was shown by an application which supports this service. Also, no message was lost.

Testing Example

An example is a network of three different Igor daemons. Two of them support the same service *foo*. Sending ten messages for the service *foo* to different IDs results in the following output at the three Igor daemons:

First node (supports service)

```
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar2'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar6'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar9'
```

Second node (supports service)

```
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar0'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar1'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar3'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar4'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar5'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar7'
the received msg has len 40 and says 'Test-Nachricht:  service fooo, Node bar8'
```

Third node

This node does not receive any message because it does not support the service *foo*.

Chapter 5

Abstract

5.1 Conclusions

With the implementation of service support into Igor it has been shown, that it is possible to extend structured overlay networks to be more useful and flexible. With the support for services it is possible to use only one network for a big number of different functions. Also, it has been shown that this can be done very efficiently and scalably. It does not matter, if only a few nodes in the network support a special service or if numerous nodes support it.

5.2 Further Work

In future more work on Igor will have to be done. The plugin concept makes it possible to implement further policies. But also work will have to be done on service related functions. It should be possible to route message not only in the *value overlay network* but also in the *attribute overlay network*. Also hidden problems could appear in big networks. For example it could happen that because of caching a *value overlay network* will be split into disjoint subnetworks. This will have to be investigated by more testing. Particularly the behaviour of the system with regard to border cases will have to be verified. The routing has to be correct, if

1. there are a large number of different services.
2. a lot of nodes support a service x, but only a few nodes supports a service y.
3. the *gateway server* for a special service is very slow
4. the *gateway server* for a service gets lost or changes often

Bibliography

[Fuh07] T. Fuhrmann. Videgor.net - der verteilte Videorekorder, 06 2007. `http://www.videgor.net/`.

[KKRO03] Michael Klein, Birgitta König-Ries, and Philipp Obreiter. Lanes – a lightweight overlay for service discovery in mobile ad hoc networks. *3rd Workshop on Applications and Services in Wireless Networks (ASWN2003). Berne, Swiss*, 2003.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, 1, 2002.

[Pub95] Federal Information Processing Standards Publications. Secure hash standard, April 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed System Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. *SIGCOMM*, pages 161–172, 2001.

[SMK+01] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 ACM Sigcomm Conference*, pages 149–160, 2001.

[SW05] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications.* Springer, Berlin, 2005.

[XYW05] Qi Xia, Ruijun Yang, and Weinong Wang. Fully decentralized DHT based approach to grid service discovery using overlay networks. *Proceedings of the 2005 The Fifth International Conference on Computer and Information Technology (CIT'05)*, 2005.